THE VLISP MODEL:

DESCRIPTION, IMPLEMENTATION, AND EVALUATION

by Jérôme Chailloux, December 1979

translated from French by Paul Y Gloess

28 January 1980

Note: The present document contains the translation of Chapter 3 (the VCMC2 machine) of Chailloux's document. Some parts of the text (programs, ...) have been omitted as they contain no French: partial inclusion signals their existence.

1. INTRODUCTION

2. REPRESENTATION OF VLISP OBJECTS

3. THE V C M C 2 MACHINE

This chapter describes the VCMC2 machine used in our work. The design and implementation of the VCMC2 machine are based on the following ideas

1. VCMC2 is a machine. The description of the implementation of our model takes into account the necessary limitations of a machine:

   - the resources (memory or registers) are limited,
   - any operation on the objects must be explicitly defined in terms of a fixed instruction set.

2. VCMC2 is a virtual machine. We do not restrict ourselves to a specific machine, especially for the internal representation of objects.

3. VCMC2 is a referential machine. The model description is portable (three systems have been built very rapidly, within a few weeks) and short (the size of the interpreter and the 160 standard functions altogether is about 2 k words [1]).

4. VCMC2 is a prototype machine. It is used as a preliminary study for the realization of a real VCMC2 machine based on a bipolar micro-processor "à tranches" [2] such as the Am2900 [AMD 78a].

3.1 ORGANIZATION OF THE MACHINE

The VCMC2 machine comprises a memory, registers and an Arithmetic and Logic Unit.

---

[1] A word is 8 bits.

[2] "Tranche" means slice.

3.1.1 The memory

The memory of the VCMC2 machine is a random-access memory. The size of its words is not fixed and depends on specific incarnations of the machine on existing materiels. A VCMC2 word must be able to contain an address. The word size will determine the maximal size of the memory (its address space)[3].

This memory is divided into several areas. Each area contains a fixed data-type.

The areas are:

1. the memory area containing the programs (the interpreter, the input/output routines, the work areas, ...). This area is not dynamically allocated. It is called the code area.

2. the stack. This memory area is managed automatically by the machine by means of specific instructions that manipulate a special pointer, the stack pointer. This stack may be viewed as an external processor with its own memory, in the spirit of the NK3 LISP machine [NAGAO79].

3. the lists area. This area is accessed by specific machine instructions and managed dynamically.

4. the numbers area. Like for the previous area, this area is managed automatically and dynamically.

5. the area for atomic symbols, which is also dynamic.

---

[3] Until now most of the machines had 16 bits addresses that addressed ed only 64 k words, which is not sufficient at all for VLISP machines. Today's trend (with the 16-bits word micro-processors) is to increase the address sizes: 20 bits for the Intel 8086 [INTEL 79] address 1 M-bytes, 23 bits for the Zilog 8000 [AMD 79] ou le Motorolla 68000 [MOTOROLLA 79] address 8 M-bytes.

The notion of memory areas may be identified with the notion of memory SEGMENTs such as those of the new 16-bits-word micro-processors, e.g., Intel/8086 [INTEL 79] or Z8000 [AMD 79b].

3.1.2 The Registers

For efficiency reasons, the VCMC2 machine has rapid memories (called registers). The number of these registers is limited to 7 in our model.

Three of them are devoted to particular tasks (PC, SP, IX) and the number of 4 general purpose registers has been chosen upon statistical data on the use of registers[4]. Moreover, a small number of registers facilitates the implementation of our machine on existing machines such as the PDP11 (8 registers), the 8080 (5 registers)[5], the PDP10, Z8000, Am29000 (16 registers).

There are 3 specialized registers, used by the machine, and 4 general purpose registers, used by the interpreter.

1. PC : this register is the program counter of the VCMC2 machine and always contains the address of the next instruction to be executed by the machine. PC is a

---

[4] The statistics given in Section 3.3.2 show that the first general purpose register is used dynamically for 37% of the operands, the second for 19% of the operands, the third for 7% and the fourth for less than 3%. Thus it is not necessary to have additional registers.

[5] The 8080 has only 5 registers that are able to contain pointers: PC, SP, HL, DE and BC. The missing registers are replaced by memory words, which allows the use of our model at the expense of efficiency.

program pointer.

2. SP : the stack pointer. It always contains the address of the last word that has been pushed. It is updated at each modification of this stack.

3. IX : this is the only index register of the machine. It is used as a pointer to the work memories of the code area.

4. A1 : is the the first general purpose register (or accumulator 1).

5. A2 : is the second general purpose register (or accumulator 2).

6. A3 : is the third general purpose register (or accumulator 3).

7. A4 : is the fourth general purpose register (or accumulator 4).

3.1.3 The Arithmetic and Logic Unit (U.A.L.)

VCMC2 relies on a universal UAL, able to perform pointer manipulations as well as type testing or arithmetic operations. Bipolar UALs such as Am2903 [AMD 78a] are perfectly suitable for the logical part, and the arithmetic unit Am9511 [AMD 78c] (although rather slow: 54-368 cycles for 32-bits flotting-point addition) allows arithmetic operations on 16 or 32 bits integers and 32-bits flotting-point numbers.

3.2 THE OBJECTS DEALT WITH AND THE OPERANDS

The VCMC2 machine will deal with pointers. A pointer is the address of a VLISP object of a certain type. This type is that of the area containing the object. Any access to the value of a VLISP object (or to the various values if the object is an atomic symbol) is performed indirectly through this pointer, in the

corresponding memory area, by means of specific instructions relevant to each value.

The VCMC2 machine uses 4 types of pointers which are :

- the type atomic symbol,
- the type number,
- the type list,
- the type binary object (which comprises anything besides VLISP objects, i.e., the stack area, and the code area which contains the instructions and work memories of the interpreter).

These types have been defined on the basis of a specific management (allocation and collection) of each memory area. The size of the pointers is not determined by VCMC2 but by each particular incarnation. Similarly, the number of available types may be augmented according to the possibilities of the materiel.

Here are the available types in an 8-types machine:

0	pointer to an atomic symbol,
1	pointer to an integer number,
2	pointer to a real number,
3	pointer to a character string,
4	pointer to a list cell,
5	pointer to a compacted list,
6	pointer to a stack entry,
7	pointer to an instruction.

Most of the instructions operate on operands of a specific type

(for example the addition requires that both operands be numbers, the CAR instruction operates on a list).

The VCMC2 machine checks the validity of types at each execution of an instruction and issues an error with interruption, in case of illegal use of a pointer, either to read or to write.

3.3 Fields and Decoding of an Instruction

An instruction describes an action to be realized by the VCMC2 machine. These instructions are stored in the code area. Each instruction is coded on one to four machine words. VCMC2 instructions are thus of variable length. This type of architecture, common to PDP/11 or Intel/8086-like computers, allows the use of explicit operands and an optimal storage management, by minimizing the space occupied by instructions.

-
****

- Here is included a description of the instruction format of the Intel 8086.

-
****

The first word of each VCMC2 instruction has a unique format, which allows a simple and fast decoding in an actual realization.

This first word is composed of 4 fields:

- the instruction-code field,
- the first-operand field (source operand),

- the second-operand field (destination operand),
- the CONTINUATION field.

We shall now describe the function of these fields.

3.3.1 The Instruction-Code Field

The instruction-code field contains the action to perform on the operands. This field is represented by the mnemonic name of the instruction. This field only is required in all instructions. The various instructions are described in Section 3.4 together with tables of static and dynamic frequency.

3.3.2 The Operand Fields

Each instruction may contain the specification of 1 or 2 operands that will be used during its execution. The first operand is called source operand and the second one destination operand, as it is usually the case for the transfer instruction. This instruction is called MOVE. It takes 2 operands. The first one is the sender (source) and the second one is the receiver (destination). The instruction transfers the contents of the sender into the receiver.

The VCMC2 machine has 8 standard operands:

1. The NIL constant.

Operand	NIL

This operand is used to specify the NIL constant (without using special atomic-symbol manipulation instructions). It must never appear as a destination operand, or it will cause a machine error. It is not possible to modify any natural property of the NIL constant, because the machine frequently uses them for its own needs and does not check their validity for efficiency reasons.

2. One of the 4 general registers (accumulators) of the machine.

Operands	A1, A2, A3 or A4

3. The top of the stack.

Operand	TST

This operand gives access to the stack. Used as a source operand, it performs a pop operation; used as a destination operand, it performs a push operation. In both cases, the stack pointer (SP) is updated to point to the most recently pushed word. In both cases the stack is modified. Stack overflow or emptiness cause a VCMC2 machine error.

4. An explicit pointer located in the word following the instruction. This pointer may be a pointer to some VLISP object or a pointer to the memory area that contains the programs (these pointers are represented by labels in the code area). This operand is called immediate operand.

Operand	'VLISP-object or (memory-address)

5. A pointer whose address is located in the word following the instruction. This operand gives direct access to memory words of the code area whose addresses are known. It is called direct operand.

Operand	(memory-address)

Here are the use frenquencies for each of the 8 types of operand of the VCMC2 machine. The static frequencies have been computed by analyzing all the VCMC2 code contained in appendices C, E and F. The dynamic frequencies have been computed on the basis of the test given in Appendix G.

Static occurrences of the operands

Number of visited operands : 2640
Number of instructions : 1320

1	NULL	= 476	18.03 %	36.06 %
2	A1	= 650	24.62 %	49.24 %
3	A2			
4	A3			
5	A4			
6	TST			
7	IMMEDIATE			
8	DIRECT			

Dynamic occurrences of operands

These data show:

- the intense use of the general registers: more than 50% statically and more than 60% dynamically. These numbers prove the usefullness of a small number of registers that save 60% of the memory accesses performed by the machine to fetch the operands.

- the decreasing use frequency of the general registers (A1, A2, A3, A4) down to 4% dynamical for A4. Thus, this number of 4 registers is well suited and additional registers would not substantially improve our model.

3.3.3 The Continuation field

Each instruction contains a fourth field, the continuation field, which specifies the address of the next instruction to be executed [6][7].

The introduction of this last field makes it possible to:

- reduce the space devoted to programs. Since all instructions have a continuation field, program control instructions become unuseful.

- realize conditional jumps by disabling or enabling the interpretation of the continuation field.

- facilitate the realization of the micro-processor that will be able to use this field directly to control the sequencer (like the Am2911 sequencer of the 2900 family [AMD78b]).

To reduce the space occupied by this field, the address of the

---

6 This field which is present in micro-machines can even be found in very high level languages, see for example the GOTO FIELD of SNOBOL 4 [GRISWOLD71].

7 The notion of continuation mentioned here is not totally disjoint from its use in denotational semantics[ROBINET79] but it differs from it by its operational character.

next instruction may be either implicitly specified by the PC register or the stack, or explicitly specified. In the latter case one must be able to specify a new address.

We have chosen four standard continuations in the VCMC2 machine:

1. the NOP continuation. It indicates sequential execution. The next instruction to be executed is the one that follows. There is no sequence rupture.

Continuation	[NOP] or no field

2. the JUMP continuation. It indicates a sequence rupture (branching, transfer). The address of the next instruction to execute is contained in the memory word following the instruction.

Continuation	[JUMP (memory-address)]

3. the CALL continuation. It indicates a sub-program call. The address of the sub-program to execute is contained in the next word after the instruction (like for the JUMP continuation). Moreover, before executing the transfer, the address of the next instruction (after the current instruction) is pushed on top of the SP-stack.

Continuation	[CALL (memory-address)]

4. the RETURN continuation. It indicates a sub-program return. The address of the next instruction to execute is contained in the top of the stack. After this address has been popped, the stack pointer (SP) is updated.

Continuation	[RETURN]

The following statistics show the occurrence frequency of each type of continuation fields in the VCMC2 machine. They are based on the tests already mentioned in the previous section.

Static occurrences of continuations

Number of continuation fields: 664
Number of instructions: 1320

1	JUMP	= 311	46.84 %	23.56 %
2	CALL	= 163	24.55 %	12.35 %
3	RETURN	= 190	28.61 %	23.56 %

Dynamic occurrences of continuations

Number of continuations: 25058
Number of executed instructions: 80202

These statistics show the intense use of these fields, statically (more than 50% of the instructions have a continuation field different from NOP) and dynamically (more than 30% of the executed instructions comprise a continuation field).

The storage savings brought by its use are very important. If the continuation field of an instruction takes <bcont> bits, and one instruction takes <binst> bits, each continuation saving one word (which would otherwise contain a transfer instruction), the total gain corresponding to the use of <ncont> continuation fields among <ninst> instructions is (in words):

gain = ((<ncont>*<binst>) - (<ninst>*<bcont>)) / <binst>.

In the 16-bits VCMC2 machine, the continuation fields are coded on 2 bits. The word gain, in the static case (1283 instructions occupying 2061 words and 648 continuations) is:

```
((664*16) - (1230*2)) / 16

      i.e. 499 words.

thus a 24% gain on 2125 words.
```

3.3.4 Representation of an Instruction

Each instruction is coded on one memory word, which may be followed by one to three pointers (one pointer for the source operand, one pointer for the destination operand and one pointer for the continuation). Hence there are 4 possible instruction formats: 1-word, 2-words, 3-words and 4-words instructions.

The following statistics show the static and dynamic frequency of each format, using the same tests as for the previous fields:

Static occurrences of the formats	
Number of instructions	: 1320
Number of words	: 2125

Dynamic occurrences of the formats

These results show that the flexibility given by variable-length instructions does not waste memory: the average space by instruction is 1.5 words.

Here are the locations of the various fields within an instruction of a 16-bits machine (each x stands for 1 bit) [8] :

instruction code	source operand	destination operand	continuation field
x x x x x x x x	x x x	x x x	x x

In this study, the VCMC2 instructions are written in an assembly-language. An instruction is represented by a line which contains:

---

[8] This size, which tremendously reduces the address space, has been retained to allow the implementation of this machine on the Intel 8080 and PDP 11 16-bits machines.

1. an optional label <et>: it is a symbolic name which identifies the instruction,

2. the mnemonic name of the instruction <opcode>. These names and their meanings are listed in Section 3.4.

3. the arguments of the instruction, i.e., the operands <source> and <dest> and the continuation <cont>, separated by commas,

4. the continuation field is enclosed in square brackets [<cont>],

5. the comments are announced by the character ; and end with the end of the line.

```
|-------------------------------------------------------
|  <et>:    <opcode>  <source> , <dest> , [ <cont> ]  |
|_______________________________________________________|
```

3.3.5 Evaluation of an instruction

An instruction is evaluated in 4 phases:

1. Computation of the address of the source operand.

2. Computation of the address of the destination operand.

3. Execution of the instruction. All instructions, in addition to their specific action, set an indicator, the condition-code CC. This indicator may have 2 values: NIL or T depending on the result of the instruction.

4. If the CC is equal to T (is true), the continuation field is analyzed and executed. If the CC is equal to NIL (is false), the continuation field is not analyzed and the NOP continuation is systematically executed (after having jumped over the operand of the continuation field in the case of a JUMP or CALL continuation).

The evaluation of the various fields is performed in a precise order: source operand, then destination operand, continuation

field at last, which removes any ambiguity in the case of several arguments.

For instance the instruction

```
+-------------------------------------+
|  <opcode>   'FOO,'BAR,[JUMP (RE)]   |
+-------------------------------------+
```

is stored in 4 words of the code area as follows:

```
+-------------+
|  <opcode>   |
|    'FOO     |
|    'BAR     |
|    (RE)     |
+-------------+
```

The machine will interpret 'FOO as the first operand, 'BAR as the second operand and (RE) as continuation address.

3.4 INSTRUCTIONS DESCRIPTION

This section describes all the VCMC2 instructions and Appendix A contains an abstract of our machine to help the reader of the various program segments that will follow.

Here are the notations we shall use to describe the instructions:

<s>	the source operand
<d>	the destination operand
->	is transfered into

<-->	is exchanged with
CC=	the new condition-code value
CVAL	the CVAL attribute of an atomic symbol
PLIST	the PLIST attribute of an atomic symbol
FTYP	the FTYP attribute of an atomic symbol
PTYP	the PTYP attribute of an atomic symbol
+	the addition
-	the difference
*	the multiplication
/	the quotient
\	the remainder
or	the inclusive logical or
and	the logical and
xor	the exclusive or
CAR	the CAR part of a list cell
CDR	the CDR part of a list cell
(x . y)	a list cell, x is the CAR part and y the CDR part
SP	the stack pointer register
(SP)	the contents of the top of the stack
IX	the index register
x[y]	an indexed operand, i.e., an operand whose address is x+y

3.4.1 The Transfer

The MOVE instruction transfers the source operand into the destination operand.

MOVE	<s> -> <d> & CC = T

This instruction is the most widely used: because of the power of the operands, all transfers are possible:

	register	->	register
or	memory	to	or memory
or	stack	<-	or stack

3.4.2 Manipulation of Atomic Symbols

The 10 following instructions give access to the natural properties of atomic symbols: C-VAL, P-LIST, F-VAL, F-TYPE and P-TYPE. There are 2 instructions for each property, one to read the property and one to write it. They allow transfers of the types:

	register	->	the
or	memory	to	symbolic-atoms
or	stack	<-	area

CVAL
SCVAL
PLIST
SPLIST
FVAL
SFVAL
FTYP
SFTYP
PTYP

SPTYP

Nota: The S prefix at the beginning of the instructions stands for SET and indicates a writing of the property.

3.4.3 Number Manipulation

These instructions perform arithmetic operations. If the type of the operands is not integer or if the instructions cause an arithmetic exception (overflow, divide by zero), a VCMC2 machine error[9] is initiated. The machine is responsible for the management of the numbers area and, in particular, for storing the resulting value.

ADD
SUB
MUL
QUO
REM
LOGOR
LOGAND
LOGXOR

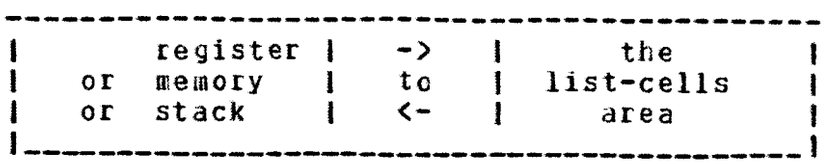
3.4.4 Manipulation of List Pointers

This set of instructions gives access to the list-cells area. This area is automatically managed by the VCMC2 machine. The

---

[9] In case of error the execution is passed to a specialized routine by means of a TRAP

garbage collector is built in the VCMC2 machine. If the garbage collector cannot function anymore (the list-area being full) a VCMC2 error is issued.



The SCAR and SCDR instructions correspond to the RPLACA and RPLACD primitive functions.

Because of the operands assymetry (NIL cannot be a destination operand, TST pushes or pops according to its position) there are 2 symetric instructions for building list-cells: CONS and XCONS, which thus give all possibilities.

```
CONS NIL,A1 -> (A1 . NIL) i.e. (A1)
CONS A1,NIL is illegal
XCONS NIL,A1 -> (NIL . A1)
```

3.4.5 Stack Usage

This instruction family provides the means of using the stack and the stack pointer SP, independently from the TST operand.

STACK
SSTACK
TOPST
XTOPST

The instructions STACK and SSTACK give read and write access to the stack pointer register itself. The TOPST instruction gives access to the top of the stack without modifying the stack nor the stack pointer. This instruction is thus used for nondestructive reading of the top of the stack. Finally, the most powerful stack-manipulation instruction is XTOPST which exchanges the top of the stack with the instruction operand without modifying the stack pointer [10].

3.4.6 IX Index-Register Usage

These instructions will use the index register either directly or to compute the actual address of the operand (in the latter case, the contents of register IX are added to the operand to yeld the address to be used).

---

[10] Even the Intel 8080 [INTEL77a] has a weakened version of this instruction: XTHL exchanges the top of the stack with the HL register.

INDEX
SINDEX
MOVEX
XMOVE

3.4.7 The Control Instructions

Owing to the utilization of the continuation field there is only a small number of these instructions. Only 2 dispatching instructions have been introduced:

JUMPX		<s>[<d>]	->	PC	&	CC= NIL
DISPT	if (LITATOM <d>)	<s>[0]	->	PC	&	CC= NIL
	if (NUMBP <d>)	<s>[1]	->	PC	&	CC= NIL
	if (LISTP <d>)	<s>[2]	->	PC	&	CC= NIL

The JUMPX instruction realizes a computed goto through a label table whose address is given by the source operand. The index in this table is provided by the destination operand (although VCMC2 considers it like a source operand).

The DISPT instruction also realizes a computed goto. The index of the label table is provided by the type of the destination operand. This instruction allows the user to test a type within a fixed amount of time, whatever the number of types is.

The following types are available:

0 atomic-symbol type

1	number type
2	list type

3.4.8 Type Testing

These istructions allow an explicit testing of the type of a pointer (atomic-symbol, number or list). They take only one operand and do not change anything besides the condition code.

TATOM
FATOM
TNUMB
FNUMB
TLIST
FLIST

Note: the T and F prefix respectively mean true and false.

3.4.9 Testing the Equality of Pointers

These two instructions test whether 2 pointers are equal or not. Two pointers are said to be equal if they point to the same physical object.

EQ	CC= (<s> = <d>)
NEQ	CC= (<s> ≠ <d>)

These instructions correspond to the EQ and NEQ VLISP predicates.

3.4.10 Arithmetic Tests

They are used to compare the values of numbers by setting the condition code.

GE
GT
LT
LE
SUBTZ
SUBFZ

The last two instructions, SUBTZ and SUBFZ, are used to implement loops.

3.4.11 The Special Instructions

A few instructions have no effect on the contents of registers or memories. These instructions perform various side effects and always set the condition code to T.

NOP	CC= T
STOP	
PRINSTACK <s>	CC= T
PRSTAT	CC= T

NOP — does not do anything besides interpreting the continuation field.

STOP — stops the VCMC2 machine.

PRINSTACK — prints the contents of the <s> last words of the stack. This instruction is used to check the stack dynamically.

PRSTAT — prints the number of executed instructions, the number of CONS realized and the maximum size of the stack since the last PRSTAT. It is used for incremental statistics purposes.

3.4.12 The Input/Output instructions

There are 2 groups of instructions: the complex input/output instructions, which work at the level of the VLISP expressions and are used in the interpreter, and the low-level instructions dealing with characters which are used by the input/output routines. These low-level instructions are described in Chapter 5 (input instructions) and 6 (output instructions).

Here are the high-level input/output instructions:

READ	input ->	<d>	CC= T
PRIN1	<s> ->	output	CC= T
TERPRI	line skip		CC= T

READ — reads a VLISP expression and stores its value into the <d> operand,

PRIN1 — prints the value of operand <s>,

TERPRI — skips one output line.

3.5 THE PSEUDO-INSTRUCTIONS

3.5.1 The MACRO's

In addition to these basic instructions, MACRO's can be used to facilitate program reading and writing.

There are 4 standard MACRO's:

PUSH <s>	is equivalent to	MOVE	<s>,TST
POP <d>	is equivalent to	MOVE	TST,<d>
TNIL <s>	is equivalent to	EQ	<s>,NIL
FNIL <s>	is equivalent to	NEQ	<s>,NIL

The PUSH and POP MACRO's are used to manipulate the stack: they are modern versions of the BURRY and UNBURRY instructions of the second Turing machine [CARPENTER76].

3.5.2 The Test Pseudo-Instructions

The VCMC2 machine includes tools for monitoring its own operation. It is possible to

- dynamically trace the instructions,
- dynamically trace the contents of the registers,
- dynamically trace the contents of the stack,
- execute a VCMC2 program in an incremental way and call the VLISP interpreter before each execution of an instruction.

These tools are described in Section 3.7.2. The following pseudo-instructions, which do not modify the contents of the memories or registers, activate or inhibit the trace or step by step mode.

TRACE	sets the trace mode
UNTRACE	cancels the trace mode
STEP	sets the step by step mode
UNSTEP	cancels the step by step mode
SILENCE	saves the current modes and

	goes into a non-trace mode
REVIVE	restores the modes saved by the last SILENCE

3.5.3 Memory-Reservation Pseudo-Instructions

There are two pseudo-instructions for reserving memory. One of them reserves a single word and the other one reserves several contiguous words.

DATA	VLISP-object or (memory-address)
BLOCK	number , VLISP-object or (memory-address)

The first one reserves one memory-word which is initialized to a VLISP-object pointer or a program-address. The second one reserves a block of contiguous words whose number is specified by the first operand and each word is initialized with the pointer specified by the second operand.

3.5.4 The Declaration Pseudo-Instructions

COMMENT	
ENTRY	<name>,<F-TYPE>,<P-TYPE>

COMMENT — is a comments declaration: the instruction remainder is ignored.

ENTRY — is an entry-point declaration for a standard function. <name> is the function name. This name must be a program label. <F-TYPE>, the

F-TYPE of the function is one of 0SUBR, 1SUBR, 2SUBR, 3SUBR, NSUBR and FSUBR. <P-TYPE>, the function P-TYPE, is a number used to control the future edition of the function. It is described in Chapter 6.

3.6 Using the VCMC2 Machine

To illustrate the use of the VCMC2 machine, we are going to code the SUBST function in VCMC2 machine-language.

The VLISP definition of this function is:

```
[1]  (DE SUBST (x y z)
```

It is a recursive function of 3 arguments which substitutes x for all occurences of y in the z list.

```
(1)          ENTRY   SUBST,3SUBR
```

This function occupies 12 VCMC2 words (usually 16-bits words)[11].

Here is the explanation of the instructions:

(1) declares the SUBST function as a SUBR of 3 arguments. This instruction corresponds to the VLISP declaration labeled [1].

---

[11] Compare this number with the 37 8-bits words required by the Peter DEUTSCH machine DEUTSCH73 to code the same function.

(2) translation of the VLISP test of line [2]. The else clause is in SUBST1:.

(3) test of line [3]: the then clause is empty because x is still in A1 and a [RETURN] can be used.

(4) else clause of the second test (i.e., z). Since the return-value of the function mus be in A1, it is necessary to transfer A3 (i.e., z) into A1 before returning from the function.

(5) else clause of the first test (i.e., CONS ...): A1 is saved because it will be destroyed by the recursive function call.

(6) save the CDR of A3 (i.e., z) into the stack. This value will be used as second argument of the CONS.

(7) first recursive call to the SUBST function: A1 contains x, A2 contains y and A3 contains (CAR z). A1 and the CDR of A3 have been pushed on the stack. When this call returns, A2 and A3 will be exchanged and A1 will contain the value of the call.

(8) restores A3 which is now ready for the second recursive call.

(9) exchanges the top of the stack (i.e., the old A1) with the value of the first recursive call and recursively calls SUBST a second time.

(10) computes the value of SUBST by setting up a list cell whose CAR is the value of the first recursive call (saved in the stack) and whose CDR is the value of the second recursive call. The constructed list cell is returned in A1 as value of the function.

3.7 MEASURING AND DEBUGGING TOOLS

The VCMC2-machine simulator comprises measuring and debugging tools for the programs written in VCMC2.

3.7.1 The Measuring Tools

The measuring tools provide the means of studying the static and dynamic aspects of a program.

The static analysis is concerned with the study of the actual VCMC2-interpreter code and the input/output routines (i.e., the code given in Appendices C, E and F). This analysis allows crossed references and various statistics such as:

- the number and the type of defined functions,
- the occurences of each type of instruction,
- the occurrences of each type of continuation field,
- the occurrence of each type of operand.

The dynamic analysis is based on the effective simulation of the instructions. This analysis returns the same kind of results as the static one, plus the following informations:

- number of CONSes performed,
- maximum size of the stack,
- time of simulation of an instruction.

All the results given in this chapter have been obtained by static or dynamic analysis.

Here is the static frequency of the instructions, for the code of the interpreter and input/output routines (i.e., the code given in Appendices C, E, F).

```
-------------------------------------------------------
|
|  Static occurrences of the instructions
|
|  Number of instructions          :  1320
```

Here are the dynamic frequencies of the instructions obtained by executing the test given in Appendix G.

```
-------------------------------------------------------
|  Dynamic occurrences of the instructions
|
|  Number of executed instructions : 80202
```

3.7.2 The Debugging Tools

The debugging tools of the VCMC2 machine consist of a trace and step by step execution.

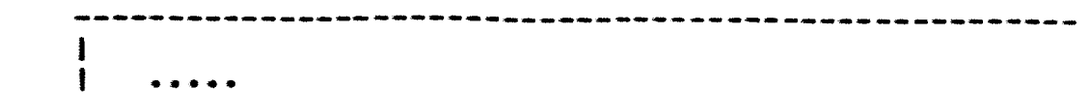
The trace eventually prints, before each execution of an instruction:

- the instruction itself,
- the contents of the registers and stack.

Here is an excerpt of the trace of the evaluation of the expression

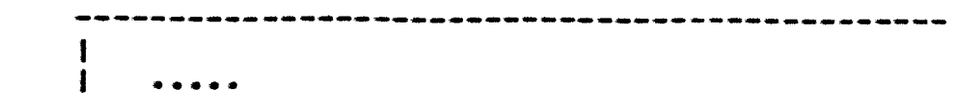
```
(SUBST 'X 'A '(A B A C))
```

using the function described in the previous section. Only the modified registers are printed by the trace:



The step by step option gives access to a TOPLEVEL VLISP before each execution of an instruction, which provides the means of consulting any register or memory, evaluating any VLISP expression or executing any VCMC2 instruction[12].

Here is an example of step by step execution of the interpreter during the evaluation of the previous expression. Before each instruction, the VCMC2 machine prints the ^ character. If the user enters a space, the instruction is executed normally. Otherwise VCMC2 evaluates any VLISP expression entered by the user. This evaluation gives access to all the interpreter and to all the variables of the simulator.



---

12 The possibility of access to a very high level language (here the VLISP interpreter itself) confers an unequalled power to this method whose capabilities greatly surpass those of the very famous DDT of DEC [DEC75c]. See the remarks of M. Model on interactive debugging in a complex environment [MODEL79].

3.8 THE INCARNATIONS OF THE VCMC2 MACHINE

The referential VCMC2 machine may be implemented in several ways, among which we shall discuss:

- the symbolic simulation,
- the binary interpretation,
- the compilation or macro-generation.

The symbolic simulation consists in a direct interpretation of the symbolic code of the programs written in VCMC2. This interpreter can only be achieved in a language allowing symbolic manipulations. VLISP is obviously suited. Thus VLISP 10 has been used to write the symbolic simulator given in Appendix B. The use of VLISP has resulted into a very short simulator (about 1000 lines) very quickly implemented (there is no need for a preprocessing of the symbolic code). We have included dynamic and symbolic debugging tools. On the other hand, this simulator is not very fast (approximately 5 milli-seconds to simulate the most complex instruction) and requires a large amount of storage (6k list cells for the simulator of Appendix B and 8k list cells to store the interpreter under its symbolic form given in Appendix F).

To decrease the amount of space occupied by the VCMC2 programs, it is possible to write a VCMC2 assembler that will translate VCMC2 programs into binary code. However, this new code will have to be interpreted by a specialized binary interpreter.

Finally, to reduce the simulation time, we have realized a translator which is able to macro-generate (or compile) the VCMC2 programs into other machines (PDP10, PDP11, Z80, T1600). This macro-generation can also be achieved by universal macro-generators such as the GPM of STRACHEY [STRACHEY65] or the STAGE 2 of [POOLE70], [WAITE73].

REFERENCES

Table of Contents

```
000000000111111111122222222223333333333444444444455555555556666666666777777777
123456789012345678901234567890123456789012345678901234567890123456789012345678

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-


 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-



  GGGG   L        OOO    EEEEE   SSSS   SSSS
 G       L       O   O   E      S      S
 G       L       O   O   E      S      S
 G       L       O   O   EEEE    SSS    SSS
 G  GGG  L       O   O   E          S      S
 G    G  L       O   O   E          S      S
  GGG    LLLLL    OOO    EEEEE  SSSS   SSSS



 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-

 *START* User GLOESS  Job  VLISP Seq. 7835 Date 28-Aug-80 14:59:41 Monitor SRI-


 0000000000000000000000000000000000000000000000000000000000000000000000000000
 0000000001111111111222222222233333333334444444444555555555566666666667777777
 1234567890123456789012345678901234567890123456789012345678901234567890123456
```